SAT-Based Software Certification

Sagar Chaki

February 2006

Predictable Assembly from Certifiable Components Initiative

Unlimited distribution subject to the copyright.

Technical Note
CMU/SEI-2006-TN-004

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2006 by Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

Abstract . . . v

1 Introduction . . . 1

2 Related Work . . . 3

3 Preliminaries . . . 4

4 Temporal Logic Witness . . . 7

5 SAT-Based Certificates . . . 18

6 Simulation . . . 20

7 Experimental Results . . . 27

8 Conclusion . . . 29

References . . . 30

List of Figures

Figure 1: Example Graph G and Induced Directed Acyclic Graph G^{SCC} 16
Figure 2: Example Graphs and LTSs Constructed from Them 25
Figure 3: Comparison of CVC, VAMPYRE, and SAT-Based Proof Generation 27

Abstract

This report formalizes a notion of witnesses as the basis of certifying the correctness of software. The first part of the report is concerned with witnesses for the satisfaction of linear temporal logic specifications by infinite state programs and shows how such witnesses may be constructed via predicate abstraction and validated by generating and proving verification conditions. In addition, the first part of this report proposes the use of theorem provers based on Boolean propositional satisfiability (SAT) and resolution proofs in validating these verification conditions. In addition to yielding extremely compact proofs, a SAT-based approach overcomes several limitations of conventional theorem provers when applied to the verification of programs written in real-life programming languages.

The second part of this report formalizes a notion of witnesses of simulation conformance between infinite state programs and finite state machine specifications. The report also proves that computing a minimal simulation relation between two finite state machines is an NP-hard problem. Finally, the report presents algorithms to construct simulation witnesses of minimal size by solving pseudo-Boolean constraints. The author's experiments on several nontrivial benchmarks suggest that a SAT-based approach can yield extremely compact proofs—in some cases by a factor of over 10^5—when compared to existing non-SAT-based theorem provers.

1 Introduction

There is an evident and urgent need for objective measures of confidence in the *behavior* of software obtained from untrusted sources. In general, the lack of trust in a piece of code stems from two sources: (1) the code producer and (2) the delivery mechanism of the code to the consumer. Unfortunately, the vast majority of current software assurance techniques target the above sources of mistrust in isolation but fail to account for them both.

For instance, cryptographic techniques are typically unable to say anything substantial about the runtime behavior of the program. Techniques such as sandboxing and analytic redundancy require mechanisms for runtime monitoring and appropriate responses to failure. Additionally, such approaches are inherently dynamic and unable to provide adequate levels of static correctness guarantees. Extrinsic software quality standards typically have a heavy focus on process and are usually quite subjective. Moreover, software qualities are weakly related to desired behavior, if at all.

This report presents a technique that uses *proofs* to certify software. More specifically, we certify a rich set of safety and *liveness* policies on C source code. Our approach consists of two broad stages. We first use model checking [Clarke 00, Clarke 82] in conjunction with Counterexample-Guided Abstraction Refinement (CEGAR) [Clarke 03] and predicate abstraction [Graf 97] to verify that a C program $\mathcal{C}$ satisfies a policy $\mathcal{S}$. The policy $\mathcal{S}$ may be expressed either as a linear temporal logic (LTL) formula or a finite state machine.

Subsequently, we use information generated by the verification procedure to extract a witness Ω. We show how the witness may be used to generate a verification condition VC. We also prove that $\mathcal{C}$ respects the policy $\mathcal{S}$ iff VC is valid. The witness Ω is constructed and shipped by the code producer along with $\mathcal{C}$ and the proof P of VC. The code consumer uses Ω to reconstruct VC and verify that P truly corresponds to VC. Therefore, in our setting, the witness Ω and the proof P may together be viewed as the certificate that $\mathcal{C}$ respects $\mathcal{S}$.

While the above strategy is theoretically sound, it must overcome two key pragmatic obstacles. First, since certificates have to be transmitted and verified, they must be small and efficiently checkable. Unfortunately, proofs generated by conventional theorem provers, such as CVC and VAMPYRE, are often prohibitively large. Second, conventional theorem provers are usually unfaithful to the semantics of C. For example, they often do not support features of integer operations such as overflow and underflow. This lack of such supoort means that certificates generated by such theorem provers are, in general, not trustworthy. For example, the following VC is declared valid by most conventional theorem provers, including CVC and VAMPYRE: $\forall x \centerdot (x + 1) > x$. However, that statement is actually invalid according to the semantics of the C language, due to the possibility of overflow.

In this report, we propose the use of Boolean satisfiability (SAT) to solve both these problems. More specifically, we translate VC to a propositional formula Φ such that VC is valid iff Φ is unsatisfiable. Therefore, a resolution refutation (proof of the unsatisfiability) of Φ serves as a proof of the validity of VC. We use the state-of-the-art SAT solver ZCHAFF [Moskewicz 01], which also generates resolution refutations, to prove that Φ is unsatisfiable. The translation from VC to Φ is faithful to the semantics of C and therefore handles issues such as overflow.

We have implemented our proposed technique in the ComFoRT [Chaki 05b] reasoning framework and experimented with several nontrivial benchmarks. Our results indicate that the use of SAT leads to extremely compact (in some cases over 10^5 times smaller) proofs in comparison to conventional theorem provers. One important reason for this improvement is that the SAT formulas generated have extremely small UNSAT-cores (i.e., subformulas that are themselves unsatisfiable). ZCHAFF has sophisticated heuristics to locate small UNSAT-cores of its input formula. Since the core is small, so is its refutation. Further details of our experiments are provided in Section 7.

We believe that this report contributes not just to the area of software certification but to the much broader spectrum of scientific disciplines where compact proofs are desirable. Algorithms to compress proof representations are currently a topic of active research. This report demonstrates that the use of SAT technology is a very promising idea in this context.

The rest of this report is organized as follows. We discuss related work in Section 2 and present preliminary concepts in Section 3. In Section 4, we present our certification formalism for LTL policies, and in Section 5, we describe our technique for obtaining SAT-based certificates. In Section 6, we present our certification formalism for finite state machine policies. Finally, we describe our experimental results in Section 7 and conclude our ideas in Section 8.

2 Related Work

Necula and Lee [Necula 96, Necula 97, Necula 98b] proposed Proof-Carrying Code (PCC) as a means for checkably certifying that untrusted *binaries* respect certain fundamental safety (such as memory safety) criteria. Foundational PCC [Appel 01, Hamid 02] attempts to reduce the trusted computing base of PCC to solely the foundations of mathematical logic. Bernard and Lee [Bernard 02] propose a new temporal logic to express PCC policies for machine code. Non-SAT-based techniques for minimizing PCC proof sizes [Necula 98a, Necula 01] and formalizing machine code semantics [Michael 00] have also been proposed. Our work uses proofs to certify software but is applicable to safety as well as liveness specifications and at the *source code* level.

Arons and colleagues [Arons 01] have proposed techniques to heuristically (and automatically) lift an invariant for a small instance of a parameterized system to a candidate invariant for the entire system. The candidate invariant is then checked for validity since, unlike in our framework, it is not known whether the smaller instance of the parameterized system is an abstraction for the full instance.

Certifying model checkers [Namjoshi 01, Kupferman 04] emit an independently checkable certificate of correctness when a temporal logic formula is found to be satisfiable by a *finite state* model. Namjoshi [Namjoshi 03] has proposed a two-step technique for obtaining proofs of μ-calculus specifications on *infinite state* systems. In the first step, a proof is obtained via certifying model checking. In the second step, the proof is *lifted* via an abstraction. This approach is more general than ours as far as LTL model checking is concerned but does not handle simulation. It also does not propose the use of SAT or provide experimental validation.

Magill and colleagues[1] have proposed a two-step procedure for certifying simulation conformance between an infinite state system and a finite state machine specification. In the first step, they certify that a finite state abstraction simulates the infinite state system. In the second step, they prove simulation between the finite state abstraction and the specification. Their approach does not cover LTL specifications and, in particular, is unable to handle liveness policies. Also, it does not propose the use of SAT.

Predicate abstraction [Graf 97] in combination with CEGAR [Clarke 03] has been applied successfully by several software model checkers such as SLAM [Ball 01], BLAST [Henzinger 02b], and MAGIC [Chaki 04a]. Out of these model checkers, SLAM and MAGIC do not generate any proof certificates when claiming the validity of program specifications. BLAST includes a method [Henzinger 02a] for lifting linear time safety proofs through the abstraction computed by their algorithm into a checkable proof of correctness for the original program. It does not handle liveness specifications and uses the non-SAT-based theorem prover VAMPYRE for proof generation. The use of SAT for software model checking has also been explored in the context of both sequential ANSI-C programs [Clarke 04] and asynchronous concurrent Boolean programs [Cook 05a]. Proving program termination via ranking functions is also a rich and developing research area [Cook 05b, Balaban 05].

[1] Magill, S.; Nanevski, A.; Clarke, E.; & Lee, P. *Simulation-Based Safety Proofs by MAGIC*. In preparation.

3 Preliminaries

In this section, we present preliminary definitions and results. Let Act be a denumerable set of actions. We begin with the notion of labeled transition systems.

Definition 1 (LTS). *A Labeled Transition System (LTS) is a quadruple* $(S, Init, \Sigma, T)$ *where: (1)* S *is a finite set of states, (2)* $Init \subseteq S$ *is a set of initial states, (3)* $\Sigma \subseteq Act$ *is a finite alphabet, and (4)* $T \subseteq S \times \Sigma \times S$ *is a transition relation.*

Given an LTS $M = (S, Init, \Sigma, T)$, we write $s \xrightarrow{\alpha} s'$ to mean $(s, \alpha, s') \in T$. Also, for any $s \in S$, and any $\alpha \in \Sigma$ we denote by $Succ(s, \alpha)$ the set of successors of s under α—in other words,

$$Succ(s, \alpha) = \{s' \mid s \xrightarrow{\alpha} s'\}$$

Linear Temporal Logic. We now define our notion of LTL. Unlike standard practice, the flavor of LTL we use is based on actions instead of propositions. This distinction is, however, inessential as far as this report is concerned. The syntax of LTL is defined by the following grammar in Backus-Naur form (where $\alpha \in Act$):

$$\phi := \alpha \mid \neg\phi_1 \mid \phi_1 \wedge \phi_2 \mid \mathbf{X}\phi_1 \mid \phi_1 \mathbf{U} \phi_2$$

The semantics of LTL is fairly standard, and we do not describe it here. In fact, we do not deal with LTL specifications directly but rather via an equivalent automata-theoretic formalism called Büchi automata.

Definition 2 (Büchi Automaton). *A Büchi automaton (or simply an automaton) is 5-tuple* $(S, Init, \Sigma, T, F)$*, where (1)* S *is a finite set of states, (2)* $Init \subseteq S$ *is a set of initial states, (3)* $\Sigma \subseteq Act$ *is a finite alphabet, (4)* $T \subseteq S \times \Sigma \times S$ *is a transition relation, and (5)* $F \subseteq S$ *is a set of final (or accepting) states.*

As in the case of LTSs, given a Büchi automaton $B = (S, Init, \Sigma, T, F)$, we write $s \xrightarrow{\alpha} s'$ to mean $(s, \alpha, s') \in T$. Also, for any $s \in S$ and any $\alpha \in \Sigma$, we denote by $Succ(s, \alpha)$ the set $\{s' \mid s \xrightarrow{\alpha} s'\}$. A *trace* $t \in Act^{\omega}$ is an infinite sequence of actions. The language accepted by an automaton is a set of traces defined as follows.

Definition 3 (Language). *Let* $B = (S, Init, \Sigma, T, F)$ *be any automaton and* $t = \langle \alpha_0, \alpha_1, \ldots \rangle$ *be any trace. A* run r *of* B *on* t *is an infinite sequence of states* $\langle s_0, s_1, \ldots \rangle$ *such that (1)* $s_0 \in Init$ *and (2)* $\forall i \geq 0 \centerdot s_i \xrightarrow{\alpha_i} s_{i+1}$*. For any run* r*, we write* $Inf(r)$ *to denote the set of states appearing infinitely often in* r*. Then, a trace* t *is accepted by* B *iff there exists a run* r *of* B *on* t *such that* $Inf(r) \cap F \neq \emptyset$*. The language of* B*, denoted by* $\mathcal{L}(B)$*, is the set of traces accepted by* B*.*

We define the product between an LTS and an automaton in the standard manner as follows:

Definition 4 (Product Automaton). *Let $M = (S_1, Init_1, \Sigma_1, T_1)$ be an LTS and $B = (S_2, Init_2, \Sigma_2, T_2, F_2)$ be an automaton such that $\Sigma_1 = \Sigma_2$. Then, the product of M and B is denoted by $M \otimes B$ and defined as the automaton $(S, Init, \Sigma, T, F)$ where (1) $S = S_1 \times S_2$, (2) $Init = Init_1 \times Init_2$, (3) $\Sigma = \Sigma_1$, (4) $F = S_1 \times F_2$, and (5) T is defined as follows:*

$$\forall s_1, s_1' \in S_1 \centerdot \forall s_2, s_2' \in S_2 \centerdot \forall \alpha \in \Sigma \centerdot (s_1, s_2) \xrightarrow{\alpha} (s_1', s_2') \iff s_1 \xrightarrow{\alpha} s_1' \wedge s_2 \xrightarrow{\alpha} s_2'$$

Program. We have applied our ideas to actual C programs. However, for clarity and simplicity of presentation, we use a programming language based on guarded commands. Let *Var* be a denumerable set of integer variables. The set of expressions *Expr* is defined over *Var* using the operators $+, -, \times, \div, =, <, \neg, \wedge$, and the C bit-wise operators.

Program Syntax. An assignment is a pair (v, e) where $v \in Var$ denotes the left-hand side (LHS) and $e \in Expr$ denotes the right-hand side (RHS). The set of assignments is denoted by *Asgn*. A guarded command is a triple (Grd, Evt, Cmd) where $Grd \in Expr$ is a guard, $Evt \in Act$ is an event, and $Cmd \in Asgn$ is an assignment. The set of guarded commands is denoted by *GrdCmd*. Given a guarded command $gc = (g, e, c)$, we write $Grd(gc)$, $Evt(gc)$, and $Cmd(gc)$ to denote g, e, and c respectively. Finally, a program is a pair (I, C) where $I \in Expr$ expresses constraints on the initial states of the program and $C \subseteq GrdCmd$ is a finite set of guarded commands.

Store. A *store* is a function $\sigma : Var \to \mathbb{Z}$ from variables to integers. The set of all stores is denoted by *Sto*. Any store σ naturally induces a function from expressions to integers: $\sigma(e)$ is the integer obtained by evaluating e under σ.

Our language has a C-like semantics as far as variables and operators are concerned. Integers are treated as 32-bit vectors. Also, the arithmetic, relational, Boolean, and bit-wise operators are interpreted in a C-like manner. In particular, there is overflow and underflow, and zero is treated as FALSE, while all other integers are treated as TRUE.

Definition 5 (Store Update). *Given a store σ and an assignment $a = (v, e)$, we write $a[\sigma]$ to denote the store resulting after executing a from σ. In other words, $a[\sigma]$ is the same as σ for all variables other than v, while $a[\sigma](v) = \sigma(e)$.*

Definition 6 (Satisfaction). *Given a store σ and an expression e, we say that σ satisfies e iff $\sigma(e) \neq 0$. We denote the satisfaction of e by σ as $\sigma \models e$ and write $\sigma \not\models e$ to mean $\neg(\sigma \models e)$.*

In the rest of this report, we use the terms *formula* and *expression* synonymously, since, as we have seen, any expression e can also be viewed as a logical formula. The models of e are simply the stores satisfying e.

Program Semantics. We now define the semantics of a program *Prog* in terms of a labeled transition system. Intuitively, the states of the LTS are stores, its initial states are determined by the initial condition of *Prog*, and its transitions are determined by the guarded commands in *Prog*. Formally, let $Prog = (I, C)$ be a program. Then, the semantics of *Prog*, denoted by $[\![Prog]\!]$, is an LTS $(S, Init, \Sigma, T)$ such that

1. $S = Sto$
2. $Init = \{\sigma \mid \sigma \models I\}$
3. $\Sigma = \{Evt(gc) \mid gc \in C\}$
4. $\sigma \xrightarrow{\alpha} \sigma'$ iff: $\exists gc \in C \centerdot \sigma \models Grd(gc) \wedge \alpha = Evt(gc) \wedge \sigma' = Cmd(gc)[\sigma]$

Specification Satisfaction. Given a specification as a *negated* automaton *Spec*, we say that *Prog* satisfies *Spec* and denote this by $Prog \models Spec$, iff $\mathcal{L}([\![Prog]\!] \otimes Spec) = \emptyset$.

4 Temporal Logic Witness

In this section, we present our proof framework for programs. We consider a program $Prog = (I, C)$. We begin with the notion of strongest postconditions. For any expression e, variable v, and expression t, we denote the expression obtained by simultaneously replacing all occurrences of v in e by t as $e[v/t]$.

Definition 7 (Strongest Postcondition). *Let $Prog = (I, C)$ be a program, e be an expression, and α be an action. Then, the strongest postcondition of e with respect to α is denoted by $\mathcal{SP}[e]\{\alpha\}$ and defined as follows:*

$$\mathcal{SP}[e]\{\alpha\} = \exists v' \centerdot \bigvee_{(g,\alpha,(v,t)) \in C} (g \wedge e)[v/v'] \wedge (v = t[v/v'])$$

The concept of strongest postconditions is quite standard. In particular, the following fact about strongest postconditions is fairly well-known. Recall that a state of $Prog$ is a store. Consider any expression e and any action α. Let σ and σ' be stores such that $\sigma \models e$ and $\sigma \xrightarrow{\alpha} \sigma'$. Then, $\sigma' \models \mathcal{SP}[e]\{\alpha\}$. This idea is captured by the following well-known fact.

Fact 1 *Let $Prog$ be a program and $[\![Prog]\!] = (S, Init, \Sigma, T)$ be its semantics. Let e be any expression. Then, the following holds:*

$$\forall \sigma \in S \centerdot \forall \sigma' \in S \centerdot \forall \alpha \in \Sigma \centerdot ((\sigma \models e) \wedge (\sigma \xrightarrow{\alpha} \sigma')) \Rightarrow (\sigma' \models \mathcal{SP}[e]\{\alpha\})$$

In addition, the following lemma about strongest postconditions will be useful later on.

Lemma 1 *Let e_1, e_2 be any expressions and α be any action. Then, the following holds:*

$$(\mathcal{SP}[e_1]\{\alpha\} \vee \mathcal{SP}[e_2]\{\alpha\}) \iff \mathcal{SP}[e_1 \vee e_2]\{\alpha\}$$

Proof.

$$\mathcal{SP}[e_1]\{\alpha\} \vee \mathcal{SP}[e_2]\{\alpha\}$$

$$\iff$$

$$\exists v' \centerdot \bigvee_{(g,\alpha,(v,t)) \in C} (g \wedge e_1)[v/v'] \wedge (v = t[v/v'])$$

$$\bigvee$$

$$\exists v' \centerdot \bigvee_{(g,\alpha,(v,t)) \in C} (g \wedge e_2)[v/v'] \wedge (v = t[v/v'])$$

$$\iff$$

$$\exists v' \centerdot \bigvee_{(g,\alpha,(v,t)) \in C} (g \wedge (e_1 \vee e_2))[v/v'] \wedge (v = t[v/v'])$$

$$\iff$$

$$\mathcal{SP}[e_1 \vee e_2]\{\alpha\}$$

This completes our proof.

$\square$

We are now ready to present the formal notion of a proof of *Prog* $\models$ *Spec*. Recall that our goal is to prove $\mathcal{L}([\![Prog]\!] \otimes Spec) = \emptyset$. Such a proof essentially encodes a *stratified ranking function* between $[\![Prog]\!]$ and *Spec*. Let us write $M_\otimes$ to mean $[\![Prog]\!] \otimes Spec$. Let $M_\otimes = (S_\otimes, Init_\otimes, \Sigma, T_\otimes, F_\otimes)$ and R be a finite set of integral ranks. Suppose that there exists a ranking function $\rho : S_\otimes \to R$ such that the following holds:

- **(RANK1)** $Init_\otimes \subseteq Domain(\rho)$, that is, all initial states of $M_\otimes$ have a rank.
- **(RANK2)** $\forall s \xrightarrow{\alpha} s' \centerdot s \notin F_\otimes \Rightarrow \rho(s) \geq \rho(s')$
- **(RANK3)** $\forall s \xrightarrow{\alpha} s' \centerdot s \in F_\otimes \Rightarrow \rho(s) > \rho(s')$

Then, there is no infinite path of $M_\otimes$ that visits an accepting state infinitely often, that is, $\mathcal{L}(M_\otimes) = \emptyset$. We use a witness to encode a ranking function. We also use appropriate side-conditions to ensure that the ranking function satisfies the three conditions mentioned above. We now state this formally:

Theorem 1 (LTL Witness). *Let Prog* $= (I, C)$ *be a program and Spec* $= (S, Init, \Sigma, T, F)$ *be a specification automaton. Let* R *be a finite set of integral ranks. Suppose that there exists a function* $\Omega : S \times R \to Expr$ *that satisfies the following four conditions:*

1. **(C1)** $\forall s \in S \centerdot \forall r \in R \centerdot \forall r' \in R \centerdot r \neq r' \Rightarrow \neg(\Omega(s, r) \wedge \Omega(s, r'))$
2. **(C2)** $\forall s \in Init \centerdot I \Rightarrow \bigvee_{r \in R} \Omega(s, r)$
3. **(C3)** $\forall s \in S \backslash F \centerdot \forall \alpha \in \Sigma \centerdot \forall r \in R \centerdot \forall s' \in Succ(s, \alpha) \centerdot \mathcal{SP}[\Omega(s, r)]\{\alpha\} \Rightarrow \bigvee_{r' \leq r} \Omega(s', r')$
4. **(C4)** $\forall s \in F \centerdot \forall \alpha \in \Sigma \centerdot \forall r \in R \centerdot \forall s' \in Succ(s, \alpha) \centerdot \mathcal{SP}[\Omega(s, r)]\{\alpha\} \Rightarrow \bigvee_{r' < r} \Omega(s', r')$

Then, $[\![Prog]\!] \models Spec$ *and we say that* Ω *is a witness to* $[\![Prog]\!] \models Spec$.

Proof. Let us write $M_\otimes$ to mean $[\![Prog]\!] \otimes Spec$. Recall that the states of $[\![Prog]\!]$ are stores, and hence the set of states of $M_\otimes$ is $Sto \times S$. Thus, it suffices to define a ranking function $\rho : Sto \times S \to R$ that satisfies conditions **RANK1–RANK3** given above. Consider any store σ and any specification state s. Due to condition **C1**, there can be at most one rank r such that $\sigma \models \Omega(s, r)$. If such an r exists, we define $\rho(\sigma, s) = r$; else, $\rho(\sigma, s)$ is undefined.

To show that ρ satisfies condition **RANK1**, consider any initial state (σ, s) of $M_\otimes$. Recall that $Prog = (I, C)$. Hence $\sigma \models I$ and $s \in Init$. By condition **C2**, there exists $r \in R$ such that $\sigma \models \Omega(s, r)$. Therefore, $\rho(\sigma, s) = r$, which is what we want.

To show that ρ satisfies condition **RANK2**, consider any transition $(\sigma, s) \xrightarrow{\alpha} (\sigma', s')$ of $M_\otimes$ such that (σ, s) is not an accepting state of $M_\otimes$. According to Definition 4, this means $\sigma \xrightarrow{\alpha} \sigma'$, $s \xrightarrow{\alpha} s'$, and $s \notin F$. Let $\rho(\sigma, s) = r$. From the definition of ρ, this means that

$\sigma \models \Omega(s, r)$. Hence from Fact 1, we know that $\sigma' \models \mathcal{SP}[\Omega(s, r)]\{\alpha\}$. Thus, from condition **C3**, we know that there exists $r' \leq r$ such that $\sigma' \models \Omega(s', r')$. Therefore, by the definition of ρ, we have $\rho(\sigma', s') = r' \leq r = \rho(\sigma, s)$, which is again what we want.

To show that ρ satisfies condition **RANK3**, consider any transition $(\sigma, s) \xrightarrow{\alpha} (\sigma', s')$ of $M_{\otimes}$ such that (σ, s) is an accepting state of $M_{\otimes}$. According to Definition 4, this means $\sigma \xrightarrow{\alpha} \sigma'$, $s \xrightarrow{\alpha} s'$, and $s \in F$. Let $\rho(\sigma, s) = r$. From the definition of ρ, this means that $\sigma \models \Omega(s, r)$. Hence from Fact 1, we know that $\sigma' \models \mathcal{SP}[\Omega(s, r)]\{\alpha\}$. Thus, from condition **C4**, we know that there exists $r' < r$ such that $\sigma' \models \Omega(s', r')$. Therefore, by the definition of ρ, we have $\rho(\sigma', s') = r' < r = \rho(\sigma, s)$, which completes the proof. □

Suppose we are given *Prog*, $Spec = (S, Init, \Sigma, T)$ and a candidate witness Ω over a set of ranks R. Since S, Σ, and R are all finite, it is straightforward to generate a formula equivalent to the conditions **C1–C4** enumerated in Theorem 1. We call such a formula our *verification condition* and denote it by $VC(Prog, Spec, \Omega)$. In essence, on account of Theorem 1, a valid proof of $VC(Prog, Spec, \Omega)$ is also a valid proof of $Prog \models Spec$.

Theorem 1 is useful in checking the validity of a proposed witness Ω. However, it yields no technique to construct such a Ω. In this section, we present a procedure called predicate abstraction. In the next section, we show how to construct a valid witness using predicate abstraction. More specifically, if our procedure actually results in a witness Ω, then Ω is guaranteed to be valid. In other words, the verification condition $VC(Prog, Spec, \Omega)$ is guaranteed to be a valid formula. We begin with some preliminary definitions.

Definition 8 (Predicate). *A predicate is simply an expression. Let $\mathcal{P}$ be a finite set of predicates. A valuation of $\mathcal{P}$ is a function from $\mathcal{P}$ to $\{\text{TRUE}, \text{FALSE}\}$. The set of all valuations of $\mathcal{P}$ is denoted by $\mathcal{V}(\mathcal{P})$. Given a valuation $V \in \mathcal{V}(\mathcal{P})$ of $\mathcal{P}$, the concretization of $\mathcal{P}$ with respect to V is denoted by $\gamma^{\mathcal{P}}(V)$ and is the expression defined as follows:*
$\gamma^{\mathcal{P}}(V) = \bigwedge_{p \in \mathcal{P}} p^{V(p)}$, where for any predicate p, we have $p^{\text{TRUE}} = p$ and $p^{\text{FALSE}} = \neg p$.

In this report, we only consider finite sets of predicates. We write $\gamma(V)$ to mean $\gamma^{\mathcal{P}}(V)$ when $\mathcal{P}$ is clear from the context. The notion of concretization presented above means that any valuation V can also be thought of as the expression $\gamma(V)$ and, therefore, leads naturally to the notion of consistency between valuations and expressions and between two valuations.

Definition 9 (Consistency). *Let V be a valuation of a set of predicates $\mathcal{P}$ and e be an expression. We say that V is consistent with e and denote this by $V \Vdash e$, iff the expression $\gamma(V) \Rightarrow \neg e$ is invalid. In other words, $V \Vdash e \iff \exists \sigma \in Sto \centerdot \sigma \models \gamma(V) \wedge \sigma \models e$. Equivalently, $\neg(V \Vdash e)$ iff the expression $\gamma(V) \Rightarrow \neg e$ is valid.*

Consistency essentially means that a valuation and an expression are not mutually exclusive. We now define the term *weakest precondition*, a concept closely related to the term *strongest postcondition*. Recall that for any expression e, variable v, and expression t, we denote the expression obtained by simultaneously replacing all occurrences of v in e by t as $e[v/t]$.

Definition 10 (Weakest Precondition). *Let Prog* $= (I, C)$ *be a program,* e *be an expression, and* α *be an action. Then, the weakest precondition of* e *with respect to* α *is denoted by* $\mathcal{WP}[e]\{\alpha\}$ *and defined as*

$$\mathcal{WP}[e]\{\alpha\} = \bigvee_{(g,\alpha,(v,t))\in C} g \wedge e[v/t]$$

The relationship between the strongest postconditions and weakest preconditions is expressed formally by the following lemma.

Lemma 2 (Preconditions and Postconditions). *Let* e, e' *be expressions and* α *be an action. Then, the following holds:*

$$(e \Rightarrow \neg\mathcal{WP}[e']\{\alpha\}) \Rightarrow (\mathcal{SP}[e]\{\alpha\} \Rightarrow \neg e')$$

Proof. Let us begin with the assumption and prove the conclusion

$$e \Rightarrow \neg\mathcal{WP}[e']\{\alpha\} \tag{4.1}$$

Expanding out the definition of $\mathcal{WP}[e']\{\alpha\}$ in (1), we have

$$e \Rightarrow \neg \bigvee_{(g,\alpha,(v,t))\in C} g \wedge e'[v/t] \tag{4.2}$$

Pushing negation inside from (2), we have

$$e \Rightarrow \bigwedge_{(g,\alpha,(v,t))\in C} \neg g \vee \neg e'[v/t] \tag{4.3}$$

Hence from (3), for each $(g, \alpha, (v, t)) \in C$, we have

$$e \Rightarrow \neg g \vee \neg e'[v/t] \tag{4.4}$$

Applying various proof rules on (4) gives us

$$\neg e \vee \neg g \vee \neg e'[v/t] \tag{4.5}$$

Let v' be a completely fresh variable. Then, we have

$$\forall v' \centerdot \neg e[v/v'] \vee \neg g[v/v'] \vee \neg e'[v/t[v/v']] \tag{4.6}$$

$$\forall v' \centerdot \neg e[v/v'] \vee \neg g[v/v'] \vee \neg((v = t[v/v']) \wedge e') \tag{4.7}$$

$$\forall v' \centerdot \neg e[v/v'] \vee \neg g[v/v'] \vee \neg(v = t[v/v']) \vee \neg e' \tag{4.8}$$

$$\forall v' \centerdot \neg(e[v/v'] \wedge g[v/v'] \wedge (v = t[v/v'])) \vee \neg e' \tag{4.9}$$

Since (9) can be proved for each $(g, \alpha, (v, t)) \in C$, we have

$$\bigwedge_{(g,\alpha,(v,t))\in C} \forall v' \centerdot \neg(e[v/v'] \wedge g[v/v'] \wedge (v = t[v/v'])) \vee \neg e' \tag{4.10}$$

Applying various proof rules on (10) gives us

$$\bigwedge_{(g,\alpha,(v,t))\in C} \forall v' \centerdot \neg((e \wedge g)[v/v'] \wedge (v = t[v/v'])) \vee \neg e' \tag{4.11}$$

$$\forall v' \centerdot \bigwedge_{(g,\alpha,(v,t))\in C} \neg((g \wedge e)[v/v'] \wedge (v = t[v/v'])) \vee \neg e' \tag{4.12}$$

Since v' does not appear in e'

$$(\forall v' \centerdot \bigwedge_{(g,\alpha,(v,t))\in C} \neg((g \wedge e)[v/v'] \wedge (v = t[v/v']))) \vee \neg e' \tag{4.13}$$

Pulling out the negation

$$(\neg\exists v' \centerdot \bigvee_{(g,\alpha,(v,t))\in C} ((g \wedge e)[v/v'] \wedge (v = t[v/v']))) \vee \neg e' \tag{4.14}$$

$$\exists v' \centerdot \bigvee_{(g,\alpha,(v,t))\in C} ((g \wedge e)[v/v'] \wedge (v = t[v/v'])) \Rightarrow \neg e' \tag{4.15}$$

Finally, using the definition of $\mathcal{SP}[e]\{\alpha\}$ on (15), we get

$$\mathcal{SP}[e]\{\alpha\} \Rightarrow \neg e' \tag{4.16}$$

which is the desired conclusion. This completes our proof. □

We are now ready to formally define the predicate abstraction of a program with respect to a set of predicates.

Predicate Abstraction. Let $Prog = (I, C)$ be a program and $\mathcal{P}$ be a set of predicates. Let $[\![Prog]\!] = (S, Init, \Sigma, T)$ be the semantics of $Prog$. Then, the predicate abstraction of $Prog$ with respect to $\mathcal{P}$ is denoted by $\{\!\{Prog\}\!\}^{\mathcal{P}}$ and defined as an LTS $(\widehat{S}, \widehat{Init}, \widehat{\Sigma}, \widehat{T})$ where (1) $\widehat{S} = \mathcal{V}(\mathcal{P})$: the states are the valuations of $\mathcal{P}$, (2) $\widehat{Init} = \{V \in \mathcal{V}(\mathcal{P}) \mid V \Vdash I\}$, (3) $\widehat{\Sigma} = \Sigma$, and (4) $\widehat{T}$ is defined as follows:

$$\forall V, V' \in \mathcal{V}(\mathcal{P}) \centerdot \forall \alpha \in \Sigma \centerdot V \xrightarrow{\alpha} V' \iff V \Vdash \mathcal{WP}[\gamma(V')]\{\alpha\}$$

Predicate abstraction enables us to create finite LTS abstractions of our infinite state programs. More importantly, it can be automated. Given $Prog$ and $\mathcal{P}$, it is easy to construct $\{\!\{Prog\}\!\}^{\mathcal{P}}$ from the definition given above. In order to check for consistency, we use an automated theorem prover. More specifically, suppose we want to check if $V \Vdash e$. Then, in accordance with Definition 9, we check for the validity of $\gamma(V) \Rightarrow \neg e$ using a (sound) theorem prover. We assume $\neg(V \Vdash e)$ iff the theorem says that $\gamma(V) \Rightarrow \neg e$ is valid.

Generating LTL Witnesses. We now present an algorithm **WitGen** for constructing a valid witness to $[\![Prog]\!] \models Spec$. The input to **WitGen** is (1) a set of predicates $\mathcal{P}$ such that $\{\!\{Prog\}\!\}^{\mathcal{P}} \models Spec$ and (2) a ranking function ρ from the states of $\{\!\{Prog\}\!\}^{\mathcal{P}} \otimes Spec$ to a finite set of ranks R that obeys conditions **RANK1–RANK3** given in Section 4. We defer the question as to how such a set of predicates $\mathcal{P}$ and ranking function ρ may be constructed until later. The output of **WitGen** is a valid witness Ω. The following theorem conveys the key ideas behind our algorithm.

Theorem 2 (Valid Witness). *Let $Prog = (I, C)$ be a program, $Spec = (S, Init, \Sigma, T, F)$ be a finite specification automaton and $\mathcal{P}$ be a set of predicates such that $\{\!\{Prog\}\!\}^{\mathcal{P}} \models Spec$. Let $\{\!\{Prog\}\!\}^{\mathcal{P}} = (\mathcal{V}(\mathcal{P}), \widehat{Init}, \widehat{\Sigma}, \widehat{T})$. Let R be a finite set of integral ranks and $\rho : \mathcal{V}(\mathcal{P}) \times S \to R$ be a ranking function that obeys conditions* **RANK1–RANK3** *given in Section 4. Now consider the witness $\Omega : S \times R \to Expr$ defined as follows: $\Omega(s, r) = \bigvee_{V : \rho(V,s)=r} \gamma(V)$. Then, Ω is a valid witness to $[\![Prog]\!] \models Spec$.*

Proof. It suffices to show that Ω satisfies conditions **C1–C4** given in Theorem 1. We first prove **C1** by contradiction. Consider any $s \in S$ and $r, r' \in R$ such that $r \neq r'$ and $\Omega(s, r) \wedge \Omega(s, r')$ is satisfiable. Now, we know that

$$\forall V \in \mathcal{V}(\mathcal{P}) \centerdot \forall V' \in \mathcal{V}(\mathcal{P}) \centerdot V \neq V' \Rightarrow \neg(\gamma(V) \wedge \gamma(V')) \tag{4.1}$$

But this means there is some valuation $V \in \mathcal{V}(\mathcal{P})$ such that $\rho(V, s) = r \neq r' = \rho(V, s)$, which is a contradiction. This completes the proof of **C1**. For the rest of the proof, we note that the following formula is valid:

$$\bigvee_{V \in \mathcal{V}(\mathcal{P})} \gamma(V) \tag{4.2}$$

The above statement holds because it is logically equivalent to the following formula:

$$\bigwedge_{p \in \mathcal{P}} (p \vee \neg p) \tag{4.3}$$

To prove **C2**, consider any $s \in Init$ and any $V \in \widehat{Init}$. From Definition 4, we know that (V, s) is an initial state of $\{\!\{Prog\}\!\}^{\mathcal{P}} \otimes Spec$. Since ρ satisfies **RANK1**, there exists $r \in R$ such that $\rho(V, s) = r$. Hence, from the definition of Ω, we have

$$\bigvee_{V \in \widehat{Init}} \gamma(V) \Rightarrow \bigvee_{r \in R} \Omega(s, r) \tag{4.4}$$

Now, from the definition of predicate abstraction, we know that $I \Rightarrow \neg\gamma(V)$ for each $V \in \mathcal{V}(\mathcal{P}) \setminus \widehat{Init}$. Hence, the following holds:

$$I \Rightarrow \neg \bigvee_{V \in \mathcal{V}(\mathcal{P}) \setminus \widehat{Init}} \gamma(V) \tag{4.5}$$

Also, from (2), we can conclude the following:

$$\neg \bigvee_{V \in \mathcal{V}(\mathcal{P}) \setminus \widehat{Init}} \gamma(V) \Rightarrow \bigvee_{V \in \widehat{Init}} \gamma(V) \tag{4.6}$$

From (5) and (6), we know that

$$I \Rightarrow \bigvee_{V \in \widehat{Init}} \gamma(V) \tag{4.7}$$

Finally, from (4) and (7), we have

$$I \Rightarrow \bigvee_{r \in R} \Omega(s, r) \tag{4.8}$$

which is precisely **C2**. To prove **C3**, consider any $s \in S \setminus F$, any $\alpha \in \Sigma$, any $r \in R$, and any V such that $\rho(V, s) = r$. From the definition of predicate abstraction, we know that, for each $V' \in \mathcal{V}(\mathcal{P}) \setminus \mathit{Succ}(V, \alpha)$, the following holds:

$$\gamma(V) \Rightarrow \neg \mathcal{WP}[\gamma(V')]\{\alpha\} \tag{4.9}$$

Using Lemma 2, for each $V' \in \mathcal{V}(\mathcal{P}) \setminus \mathit{Succ}(V, \alpha)$, we have

$$\mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \neg \gamma(V') \tag{4.10}$$

Hence, the following holds:

$$\mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \neg \bigvee_{V' \in \mathcal{V}(\mathcal{P}) \setminus \mathit{Succ}(V,\alpha)} \gamma(V') \tag{4.11}$$

Also, from (2), we can conclude

$$\neg \bigvee_{V' \in \in \mathcal{V}(\mathcal{P}) \setminus \mathit{Succ}(V,\alpha)} \gamma(V') \Rightarrow \bigvee_{V' \in \mathit{Succ}(V,\alpha)} \gamma(V') \tag{4.12}$$

From (11) and (12), we have

$$\mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \bigvee_{V' \in \mathit{Succ}(V,\alpha)} \gamma(V') \tag{4.13}$$

Now consider any $s' \in \mathit{Succ}(s, \alpha)$. From Definition 4, we know that $\forall V' \in \mathit{Succ}(V, \alpha) \centerdot (V, s) \xrightarrow{\alpha} (V', s')$ is a transition of $\{\!\!\{ \mathit{Prog} \}\!\!\}^{\mathcal{P}} \otimes \mathit{Spec}$ and also that (V, s) is not an accepting state of $\{\!\!\{ \mathit{Prog} \}\!\!\}^{\mathcal{P}} \otimes \mathit{Spec}$. Since ρ obeys condition **RANK2**, we know that $\rho(V', s') \leq \rho(V, s) = r$. Hence, from the definition of Ω, we have

$$\gamma(V') \Rightarrow \bigvee_{r' \leq r} \Omega(s', r') \tag{4.14}$$

Since V' is an arbitrary element of $\mathit{Succ}(V, \alpha)$, from (14), we have

$$\bigvee_{V' \in \mathit{Succ}(V,\alpha)} \gamma(V') \Rightarrow \bigvee_{r' \leq r} \Omega(s', r') \tag{4.15}$$

From (13) and (15), we have

$$\mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \bigvee_{r' \leq r} \Omega(s', r') \tag{4.16}$$

Since V is any valuation such that $\rho(V, s) = r$, from (16), we have

$$\bigvee_{V : \rho(V,s) = r} \mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \bigvee_{r' \leq r} \Omega(s', r') \tag{4.17}$$

From (17) and Lemma 1, we have

$$\mathcal{SP}[\bigvee_{V : \rho(V,s) = r} \gamma(V)]\{\alpha\} \Rightarrow \bigvee_{r' \leq r} \Omega(s', r') \tag{4.18}$$

Finally from (18) and the definition of Ω, we have

$$\mathcal{SP}[\Omega(s,r)]\{\alpha\} \Rightarrow \bigvee_{r' \leq r} \Omega(s',r') \tag{4.19}$$

which is precisely **C3**. The proof of **C4** is very similar to that of **C3**. We present it here for the sake of completeness. Consider any $s \in F$, any $\alpha \in \Sigma$, any $r \in R$, and any V such that $\rho(V,s) = r$. From the definition of predicate abstraction, we know that, for each $V' \in \mathcal{V}(\mathcal{P}) \setminus \mathit{Succ}(V,\alpha)$, the following holds:

$$\gamma(V) \Rightarrow \neg\mathcal{WP}[\gamma(V')]\{\alpha\} \tag{4.20}$$

Using Lemma 2, for each $V' \in \mathcal{V}(\mathcal{P}) \setminus \mathit{Succ}(V,\alpha)$, we have

$$\mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \neg\gamma(V') \tag{4.21}$$

Hence, the following holds:

$$\mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \neg \bigvee_{V' \in \mathcal{V}(\mathcal{P}) \setminus \mathit{Succ}(V,\alpha)} \gamma(V') \tag{4.22}$$

Also, from (2), we can conclude

$$\neg \bigvee_{V' \in \in \mathcal{V}(\mathcal{P}) \setminus \mathit{Succ}(V,\alpha)} \gamma(V') \Rightarrow \bigvee_{V' \in \mathit{Succ}(V,\alpha)} \gamma(V') \tag{4.23}$$

From (22) and (23), we have

$$\mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \bigvee_{V' \in \mathit{Succ}(V,\alpha)} \gamma(V') \tag{4.24}$$

Now consider any $s' \in \mathit{Succ}(s,\alpha)$. From Definition 4, we know that $\forall V' \in \mathit{Succ}(V,\alpha) \,.\, (V,s) \xrightarrow{\alpha} (V',s')$ is a transition of $\{\!\!\{\mathit{Prog}\}\!\!\}^{\mathcal{P}} \otimes \mathit{Spec}$ and also that (V,s) is an accepting state of $\{\!\!\{\mathit{Prog}\}\!\!\}^{\mathcal{P}} \otimes \mathit{Spec}$. Since ρ obeys condition **RANK3**, we know that $\rho(V',s') < \rho(V,s) = r$. Hence, from the definition of Ω, we have

$$\gamma(V') \Rightarrow \bigvee_{r' < r} \Omega(s',r') \tag{4.25}$$

Since V' is an arbitrary element of $\mathit{Succ}(V,\alpha)$, from (25), we have

$$\bigvee_{V' \in \mathit{Succ}(V,\alpha)} \gamma(V') \Rightarrow \bigvee_{r' < r} \Omega(s',r') \tag{4.26}$$

From (24) and (26), we have

$$\mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \bigvee_{r' < r} \Omega(s',r') \tag{4.27}$$

Since V is any valuation such that $\rho(V,s) = r$, from (27), we have

$$\bigvee_{V : \rho(V,s) = r} \mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \bigvee_{r' < r} \Omega(s',r') \tag{4.28}$$

From (28) and Lemma 1, we have

$$\mathcal{SP}[\bigvee_{V:\rho(V,s)=r} \gamma(V)]\{\alpha\} \Rightarrow \bigvee_{r'<r} \Omega(s',r') \tag{4.29}$$

Finally, from (29) and the definition of Ω, we have

$$\mathcal{SP}[\Omega(s,r)]\{\alpha\} \Rightarrow \bigvee_{r'<r} \Omega(s',r') \tag{4.30}$$

which is precisely **C4**. This completes the proof. $\square$

Getting Predicates and Ranking Functions. Theorem 2 immediately leads to an algorithm **WitGen** to construct a valid witness Ω to $Prog \models Spec$. However, **WitGen** requires as input an appropriate set of predicates $\mathcal{P}$ such that $\{\!\{Prog\}\!\}^{\mathcal{P}} \models Spec$, as well as a ranking function ρ satisfying the conditions mentioned in Theorem 2. A suitable $\mathcal{P}$ may be constructed by combining predicate abstraction with CEGAR. More specifically, starting with an initially empty $\mathcal{P}$, we use the following iterative procedure:

1. Construct $\{\!\{Prog\}\!\}^{\mathcal{P}}$.
2. Check if $\{\!\{Prog\}\!\}^{\mathcal{P}} \models Spec$. If so, we are done. Otherwise, we obtain a counterexample *CE* to $\{\!\{Prog\}\!\}^{\mathcal{P}} \models Spec$.
3. Check if *CE* is a valid counterexample. If so, then $Prog \not\models Spec$. Hence, no suitable $\mathcal{P}$ exists, and we exit unsuccessfully.
4. Otherwise, we construct a new set of predicates $\mathcal{P}$ such that $\mathcal{P}$ *eliminates CE* and then go back to Step 1.

Full details of such a procedure can be found elsewhere [Chaki 04b]. Due to the fundamental undecidability of the problem, such an approach is not always guaranteed to terminate. However, CEGAR-based techniques have been reported to be quite successful [Ball 01, Henzinger 02b, Chaki 04a] in software verification in recent times.

Generating the Ranking Function. Once an appropriate set of predicates $\mathcal{P}$ has been found by the above procedure, we have to construct a ranking function ρ. More precisely, suppose that $\{\!\{Prog\}\!\}^{\mathcal{P}} = (\mathcal{V}(\mathcal{P}), \widehat{Init}, \widehat{\Sigma}, \widehat{T})$ and $Spec = (S, Init, \Sigma, T, F)$. Then, we have to construct (1) a finite set of integral ranks R and (2) a ranking function $\rho : \mathcal{V}(\mathcal{P}) \times S \rightarrow R$ that obeys conditions **RANK1–RANK3** given in Section 4. We now give an algorithm to achieve these two goals.

Let us denote $\{\!\{Prog\}\!\}^{\mathcal{P}} \otimes Spec$ by $M_\otimes$ and let $M_\otimes = (S_\otimes, Init_\otimes, \Sigma, T_\otimes, F_\otimes)$. Without loss of generality, we assume that both $S_\otimes$ and $F_\otimes$ only contain the states of $M_\otimes$ that are reachable from $Init_\otimes$ via the transition relation. Our ranking function is defined on only $S_\otimes$, and undefined for unreachable states of $M_\otimes$.

First, we note that $M_\otimes$ can be viewed as a directed graph $G_\otimes = (N, E)$ such that $(N = S_\otimes) \bigwedge (E = \{(s,s') \mid \exists \alpha \in \Sigma \centerdot s \xrightarrow{\alpha} s'\})$. Given any two nodes s and s', we say that $s \rightsquigarrow s'$ iff there is a path from s to s' in G. In other words, $s \rightsquigarrow s'$ iff there exists a finite

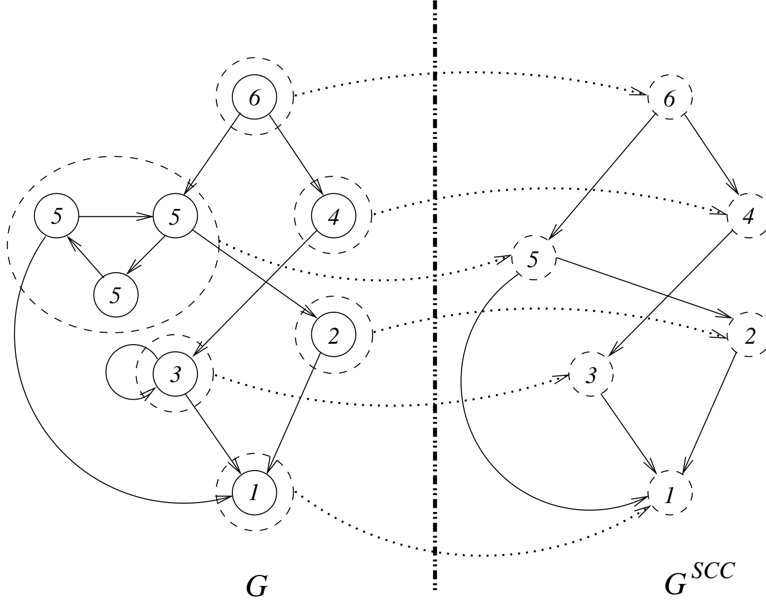

Figure 1: Example Graph G and Induced Directed Acyclic Graph G^{SCC}

non-empty sequence of states $s_1, s_2, \ldots, s_k$ such that
$(s = s_1) \bigwedge (s' = s_k) \bigwedge (\forall i \in \{1, \ldots, k-1\} \centerdot (s_i, s_{i+1}) \in E)$. A strongly connected component (SCC) of $G_\otimes$ is a set of nodes $X \subseteq N$ such that $\forall s \in X \centerdot \forall s' \in X \centerdot s \rightsquigarrow s'$. A node of $G_\otimes$ that does not belong to any SCC is called a *finitary* node. It is evident that a node n is finitary iff for every run x of $M_\otimes$ we have $n \notin \mathit{Inf}(x)$. We also know that $\{\!\!\{ \mathit{Prog} \}\!\!\}^{\mathcal{P}} \models \mathit{Spec}$ and hence $\mathcal{L}(M_\otimes) = \emptyset$. This means that every accepting state $s \in F_\otimes$ must be finitary.

It is also well-known that every directed graph G induces a directed acyclic graph G^{SCC}. The nodes of G^{SCC} are the maximal strongly connected components and the finitary nodes of G, while its edges are induced by those of G. Let $G_\otimes^{SCC}$ be the directed acyclic graph induced by $G_\otimes$. Let $\mathcal{O} = \langle n_1, n_2, \ldots, n_k \rangle$ be a topological ordering of the nodes of $G_\otimes^{SCC}$ such that if $n_i \rightsquigarrow n_j$, then n_j appears before n_i in $\mathcal{O}$. We now fix our set of ranks R to be $\{1, 2, \ldots, k\}$ where $k = |\mathcal{O}|$. We first define a ranking function ρ^{SCC} for the nodes of $G_\otimes^{SCC}$ as follows: $\rho^{SCC}(n) = i$ iff $n = n_i$ according to the ordering $\mathcal{O}$. We then use ρ^{SCC} to define a ranking function ρ for $G_\otimes$ as follows:

- If n is a finitary node, it is also a node of $G_\otimes^{SCC}$. Then $\rho(n) = \rho^{SCC}(n)$.
- Otherwise n belongs to an unique maximal SCC n^{SCC}, which is a node of $G_\otimes^{SCC}$. In this case, $\rho(n) = \rho^{SCC}(n^{SCC})$.

As an example, Figure 1 shows a G on the left and the induced G^{SCC} on the right. Each node of G is labeled by its rank inferred from a particular topological ordering.

We now show that ρ satisfies conditions **RANK1–RANK3** given in Section 4. Condition **RANK1** holds because $\mathit{Init}_\otimes \subseteq S_\otimes = \mathit{Domain}(\rho)$. For condition **RANK2**, consider any transition $s \xrightarrow{\alpha} s'$ of $M_\otimes$ such that $s \notin F_\otimes$. Now, since $s \rightsquigarrow s'$, we have $\rho(s) \geq \rho(s')$, which is

precisely **RANK2**. For condition **RANK3**, consider any transition $s \xrightarrow{\alpha} s'$ of $M_\otimes$ such that $s \in F_\otimes$. Recall that in this case, s must be a finitary node. Hence $\rho(s) \neq \rho(s')$. Since $s \rightsquigarrow s'$, we have $\rho(s) > \rho(s')$, which is precisely **RANK3**.

The use of ranking functions for proofs of liveness properties is well studied, and ours is but another instance of this methodology. The use, and limitations, of CEGAR for generating appropriate predicates are orthogonal to the witness construction procedure. In practice, any oracle capable of providing a suitable set of predicates can be substituted for CEGAR. For instance, some of the predicates can be supplied manually, and the remaining predicates may be constructed automatically.

5 SAT-Based Certificates

Suppose we are given a program *Prog*, a specification *Spec*, and a candidate witness Ω. We wish to check the validity of Ω. To this end, we construct the verification condition $VC = VC(Prog, Spec, \Omega)$ and prove that VC is valid. One way to achieve this goal is to pass VC as a query to an existing proof-generating automated theorem prover such as CVC or VAMPYRE. However, there are at least two shortcomings to this approach.

First, most theorem provers treat integers, as well as operations on integers, in a manner that is incompatible with the semantics of our programming language. For example, our language defines integers to be 32-bit vectors, and operations such as addition and multiplication are defined in accordance with two's-complement arithmetic. In contrast, for most theorem provers, integers have an infinite domain, and the operations on them are the ones we learned in primary school. An important consequence of this discrepancy is that certificates generated by conventional theorem provers may be untrustworthy for our purposes. For example, the following verification condition is declared valid by most conventional theorem provers, including CVC and VAMPYRE: $\forall x \centerdot (x+1) > x$. However, that statement is actually invalid according to our language semantics due to the possibility of overflow.

In addition, the proofs generated by such theorem provers are usually quite large (see Figure 3). We propose the use of a SAT-based proof-generating decision procedure to overcome both hurdles. Recall that the verification conditions we are required to prove are essentially expressions. Given a verification condition VC, we check its validity as follows:

1. We translate VC to a SAT formula Φ in conjunctive normal form such that VC is valid iff Φ is unsatisfiable. In essence, Φ represents the negation of VC.
2. We check for the satisfiability of Φ using a SAT solver. If Φ is found to be satisfiable, VC is invalid. Otherwise, Φ is unsatisfiable, and therefore VC is valid. In such a case, our SAT solver also emits a resolution[2] proof P that refutes Φ. We use P as the proof of validity of VC.

In our implementation, we use the CPROVER [Kroening 02] tool to perform Step 1 above. Step 2 is performed by the state-of-the-art SAT solver ZCHAFF [Moskewicz 01], which is capable of generating resolution-based refutation proofs [Zhang 03]. The ZCHAFF distribution also comes with a proof checker, which we use to verify the correctness of the proofs emitted by ZCHAFF as a sanity check. We discuss our experimental results in detail in Section 7. We note here that, in almost all cases, SAT-based proofs are over 100 times (in one case, over 10^5 times) more compact than those generated by CVC and VAMPYRE. Of course, our proofs are additionally faithful to the semantics of our programming language.

It is important to understand how our approach addresses the two shortcomings of conventional theorem provers presented at the beginning of this section. The first problem regarding language semantics is handled by the translation from VC to Φ in Step 1 above.

[2] Resolution is a sound and complete inference rule for refuting propositional formulas.

Of course, the translator itself now becomes part of our trusted computing base. However, we believe that such a decision is amply justified by the resulting benefits.

The second difficulty with large proof sizes is mitigated by the fact that a Φ generated from real-life programs and specifications often has an extremely compact resolution refutation. Intuitively, if a program is correct, it is usually so because of some simple reason. In practice, this simple reason for correctness results in Φ having a much smaller unsatisfiable core C. In essence, C is a subset of the clauses in Φ that is itself unsatisfiable. Since Φ is in CNF form, it is possible to refute Φ by simply refuting C. State-of-the-art SAT solvers, such as ZCHAFF, leverage this idea by first computing a small unsatisfiable core of the target formula and then generating a refutation for only the core. Section 7 contains more details about the kind of compression we are typically able to obtain by using the unsatisfiable core.

Finally, we note that the use of SAT guarantees trustworthiness of the generated certificate, even if we use a non-SAT-based theorem prover, such as SIMPLIFY [Nelson 80], for predicate abstraction. The trustworthiness of the generated certificate enables us to use fast, but potentially unfaithful, theorem provers during the verification stage and still remain faithful to C semantics as far as certification is concerned.

6 Simulation

While LTL allows us to reason about both safety and liveness properties, it is nevertheless restricted to a purely linear notion of time. Simulation enables us to reason about the branching time properties of programs, since it preserves all specifications in the ACTL* temporal logic.

Definition 11 (Simulation). *Let* $M_1 = (S_1, Init_1, \Sigma, T_1)$ *and* $M_2 = (S_2, Init_2, \Sigma, T_2)$ *be two LTSs. Note that* M_1 *and* M_2 *have the same alphabet. A relation* $\mathcal{R} \subseteq S_1 \times S_2$ *is said to be a simulation relation if it satisfies the following condition:* (**SIM**) $\forall s_1 \in S_1 \centerdot \forall s_1' \in S_1 \centerdot \forall s_2 \in S_2 \centerdot \forall \alpha \in \Sigma \centerdot (s_1, s_2) \in \mathcal{R} \wedge s_1 \xrightarrow{\alpha} s_1' \Rightarrow \exists s_2' \in S_2 \centerdot s_2 \xrightarrow{\alpha} s_2' \wedge (s_1', s_2') \in \mathcal{R}$. *We say that* M_1 *is simulated by* M_2 *and denote this by* $M_1 \preccurlyeq M_2$*, iff there exists a simulation relation* $\mathcal{R} \subseteq S_1 \times S_2$ *such that* $\forall s_1 \in Init_1 \centerdot \exists s_2 \in Init_2 \centerdot (s_1, s_2) \in \mathcal{R}$.

Simulation Witness. We are now ready to present the formal notion of a proof of $Prog \preccurlyeq Spec$. Such a proof essentially encodes a simulation relation between *Prog* and *Spec*. The idea is to use a mapping Ω from states of *Spec* to expressions such that for any state s of *Spec*, $\Omega(s)$ is satisfied by those states of *Prog* that are simulated by *Spec*. We now state this formally.

Theorem 3 (Simulation Witness). *Let* $Prog = (I, C)$ *be a program and* $Spec = (S, Init, \Sigma, T)$ *be a finite LTS. Suppose that there exists a function* $\Omega : S \to Expr$ *that satisfies the following two conditions:* **(D1)** $I \Rightarrow \bigvee_{s \in Init} \Omega(s)$ *and* **(D2)** $\forall s \in S \centerdot \forall \alpha \in \Sigma \centerdot \mathcal{SP}[\Omega(s)]\{\alpha\} \Rightarrow \bigvee_{s' \in Succ(s,\alpha)} \Omega(s')$. *Then,* $[\![Prog]\!] \preccurlyeq Spec$*, and we say that* Ω *is a witness to* $[\![Prog]\!] \preccurlyeq Spec$.

Proof. Recall that the states of $[\![Prog]\!]$ are stores. Consider the relation $\mathcal{R} \subseteq Sto \times S$ defined as follows : $(\sigma, s) \in \mathcal{R} \iff \sigma \models \Omega(s)$. We first show that **(GOAL1)** $\mathcal{R}$ is a simulation relation by proving that $\mathcal{R}$ satisfies condition **(SIM)** from Definition 11.

1. Let (i) $(\sigma, s) \in \mathcal{R}$ and (ii) $\sigma \xrightarrow{\alpha} \sigma'$.
2. From 1(i) and the definition of $\mathcal{R}$, we know that $\sigma \models \Omega(s)$.
3. From 1(ii), 2, and Fact 1, we know that $\sigma' \models \mathcal{SP}[\Omega(s)]\{\alpha\}$.
4. From 3 and condition **D2**, we know that

$$\exists s' \in Succ(s, \alpha) \centerdot \sigma' \models \Omega(s')$$

5. From 4 and the definition of $\mathcal{R}$, we have $(\sigma', s') \in \mathcal{R}$. This completes the proof of **GOAL1**.

Next, we prove that **(GOAL2)** for every initial state σ of $[\![Prog]\!]$, there is an initial state s of *Spec* such that $(\sigma, s) \in \mathcal{R}$. From the definition of program semantics, we know that $\sigma \models I$. Then, from condition **(D1)** above, we know that $\exists s \in Init \centerdot \sigma \models \Omega(s)$. Therefore, from the definition of $\mathcal{R}$, we have $(\sigma, s) \in \mathcal{R}$. This completes the proof of **GOAL2**. Finally, from **GOAL1**, **GOAL2**, and Definition 11, we conclude that $[\![Prog]\!] \preccurlyeq Spec$.

□

Suppose we are given *Prog*, $Spec = (S, Init, \Sigma, T)$ and a candidate witness Ω. Since both S and Σ are finite, it is straightforward to generate a formula equivalent to the conditions **D1–D2** enumerated in Theorem 3. We call such a formula our *verification condition* and denote it by $VC(Prog, Spec, \Omega)$. In essence, on account of Theorem 3, a valid proof of $VC(Prog, Spec, \Omega)$ is also a valid proof of $Prog \preccurlyeq Spec$.

Generating Simulation Witnesses. We now present an algorithm **WitGenSimul** for constructing a valid witness to $[\![Prog]\!] \preccurlyeq Spec$. The input to **WitGenSimul** is a set of predicates $\mathcal{P}$ such that $\{\!\{Prog\}\!\}^{\mathcal{P}} \preccurlyeq Spec$ and a simulation relation $\mathcal{R}$ between the states of $\{\!\{Prog\}\!\}^{\mathcal{P}}$ and the states of *Spec*. We defer the question as to how such a set of predicates $\mathcal{P}$ and simulation relation $\mathcal{R}$ may be constructed until later. The output of **WitGenSimul** is a valid witness Ω. The following theorem conveys the key ideas behind our algorithm.

Theorem 4 (Valid Witness). *Let $Prog = (I, C)$ be a program, $Spec = (S, Init, \Sigma, T)$ be a finite LTS, and $\mathcal{P}$ be a set of predicates such that $\{\!\{Prog\}\!\}^{\mathcal{P}} \preccurlyeq Spec$. Let $\{\!\{Prog\}\!\}^{\mathcal{P}} = (\mathcal{V}(\mathcal{P}), \widehat{Init}, \widehat{\Sigma}, \widehat{T})$ and $\mathcal{R} \subseteq \mathcal{V}(\mathcal{P}) \times S$ be a simulation relation such that* (**A1**) $\forall V \in \widehat{Init} \centerdot \exists s \in Init \centerdot (V, s) \in \mathcal{R}$. *Let us also define a function $\theta : S \rightarrow 2^{\mathcal{V}(\mathcal{P})}$ as follows:* (**A2**) $\forall s \in S \centerdot \theta(s) = \{V \mid (V, s) \in \mathcal{R}\}$. *Now, consider the witness $\Omega : S \rightarrow Expr$ defined as follows:* (**A3**) $\forall s \in S \centerdot \Omega(s) = \bigvee_{V \in \theta(s)} \gamma(V)$. *Then, Ω is a valid witness to $[\![Prog]\!] \preccurlyeq Spec$.*

Proof. Clearly, the following formula is valid:

$$\bigvee_{V \in \mathcal{V}(\mathcal{P})} \gamma(V) \tag{6.1}$$

This is because (1) is equivalent to the following formula:

$$\bigwedge_{p \in \mathcal{P}} (p \vee \neg p) \tag{6.2}$$

First, we show that condition **D1** of Theorem 3 holds. From **A1**, **A2**, and **A3** above, we conclude that the following is valid:

$$\bigvee_{V \in \widehat{Init}} \gamma(V) \Rightarrow \bigvee_{s \in Init} \Omega(s) \tag{6.3}$$

Now, from the definition of predicate abstraction, we know that $I \Rightarrow \neg\gamma(V)$ for each $V \in \mathcal{V}(\mathcal{P}) \setminus \widehat{Init}$. Hence, the following holds:

$$I \Rightarrow \neg \bigvee_{V \in \mathcal{V}(\mathcal{P}) \setminus \widehat{Init}} \gamma(V) \tag{6.4}$$

Also, from (1), we can conclude the following:

$$\neg \bigvee_{V \in \mathcal{V}(\mathcal{P}) \setminus \widehat{Init}} \gamma(V) \Rightarrow \bigvee_{V \in \widehat{Init}} \gamma(V) \tag{6.5}$$

From (4) and (5), we know that

$$I \Rightarrow \bigvee_{V \in \widehat{Init}} \gamma(V) \tag{6.6}$$

Finally, from (3) and (6), we have

$$I \Rightarrow \bigvee_{s \in Init} \Omega(s) \tag{6.7}$$

which is precisely **D1**. We now show that condition **D2** of Theorem 3 holds. Consider any state $s \in S$, any $V \in \theta(s)$, and any $\alpha \in \Sigma$. From the definition of predicate abstraction, we know that, for each $V' \in \mathcal{V}(\mathcal{P}) \setminus Succ(V, \alpha)$, the following holds:

$$\gamma(V) \Rightarrow \neg \mathcal{WP}[\gamma(V')]\{\alpha\} \tag{6.8}$$

Using Lemma 2, for each $V' \in \mathcal{V}(\mathcal{P}) \setminus Succ(V, \alpha)$, we have

$$\mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \neg \gamma(V') \tag{6.9}$$

Hence, the following holds:

$$\mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \neg \bigvee_{V' \in \mathcal{V}(\mathcal{P}) \setminus Succ(V,\alpha)} \gamma(V') \tag{6.10}$$

Also, from (1), we can conclude

$$\neg \bigvee_{V' \in \in \mathcal{V}(\mathcal{P}) \setminus Succ(V,\alpha)} \gamma(V') \Rightarrow \bigvee_{V' \in Succ(V,\alpha)} \gamma(V') \tag{6.11}$$

From (10) and (11), we have

$$\mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \bigvee_{V' \in Succ(V,\alpha)} \gamma(V') \tag{6.12}$$

Since $(V, s) \in \mathcal{R}$ and $\mathcal{R}$ is a simulation relation, we have

$$Succ(V, \alpha) \subseteq \bigcup_{s' \in Succ(s,\alpha)} \theta(s') \tag{6.13}$$

Hence, from (12) and (13), we know that

$$\mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \bigvee_{s' \in Succ(s,\alpha)} \bigvee_{V' \in \theta(s')} \gamma(V') \tag{6.14}$$

From (14) and the definition of Ω, we have

$$\mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \bigvee_{s' \in Succ(s,\alpha)} \Omega(s') \tag{6.15}$$

Since V is any element of $\theta(s)$, from (15), we have

$$\bigvee_{V \in \theta(s)} \mathcal{SP}[\gamma(V)]\{\alpha\} \Rightarrow \bigvee_{s' \in Succ(s,\alpha)} \Omega(s') \tag{6.16}$$

From (16) and Lemma 1, we have

$$\mathcal{SP}[\bigvee_{V \in \theta(s)} \gamma(V)]\{\alpha\} \Rightarrow \bigvee_{s' \in Succ(s,\alpha)} \Omega(s') \tag{6.17}$$

Again, from (17) and the definition of Ω, we have

$$\mathcal{SP}[\Omega(s)]\{\alpha\} \Rightarrow \bigvee_{s' \in Succ(s,\alpha)} \Omega(s') \tag{6.18}$$

which is precisely **D2**. This completes our proof. □

Getting Simulation Predicates. Theorem 4 immediately leads to an algorithm **WitGenSimul** to construct a valid witness Ω to $Prog \preccurlyeq Spec$. However, **WitGenSimul** requires as input an appropriate set of predicates $\mathcal{P}$ such that $\{\!\{Prog\}\!\}^{\mathcal{P}} \preccurlyeq Spec$. As in the case of LTL model checking, such a $\mathcal{P}$ may be constructed by combining predicate abstraction with CEGAR. More specifically, starting with an initially empty $\mathcal{P}$, we use the following iterative procedure:

1. Construct $\{\!\{Prog\}\!\}^{\mathcal{P}}$.
2. Check if $\{\!\{Prog\}\!\}^{\mathcal{P}} \preccurlyeq Spec$. If so, we are done. Otherwise, we obtain a counterexample *CE* to $\{\!\{Prog\}\!\}^{\mathcal{P}} \preccurlyeq Spec$.
3. Check if *CE* is a valid counterexample. If so, then $Prog \not\preccurlyeq Spec$. Hence, no suitable $\mathcal{P}$ exists, and we exit unsuccessfully.
4. Otherwise, we construct a new set of predicates $\mathcal{P}$ such that $\mathcal{P}$ *eliminates CE* and then go back to Step 1.

Full details of such a procedure can be found elsewhere [Chaki 05a]. As in the case of LTL, due to the fundamental undecidability of the problem, such an approach is not always guaranteed to terminate but has been found to be quite effective in practice.

Witness Minimization. It is clear from Theorem 4 that the size of witnesses and proofs generated by **WitGenSimul** is directly related to the size of the simulation relation $\mathcal{R}$ between $\{\!\{Prog\}\!\}^{\mathcal{P}}$ and *Spec*. In this section, we describe an algorithm to construct a *minimal* simulation relation between two finite LTSs, if such a relation exists. Clearly, such an algorithm can be used to construct an $\mathcal{R}$ of minimal size, which would, in turn, lead to a witness Ω of small size.

Our algorithm relies on a well-known technique [Chaki 04a] to check for simulation between finite LTSs using satisfiability for weakly negated HORNSAT formulas. More specifically, suppose we are given two finite LTSs $M_1 = (S_1, Init_1, \Sigma, T_1)$ and $M_2 = (S_2, Init_2, \Sigma, T_2)$. Then, we can construct a propositional CNF formula Ψ such that the set of variables

appearing in Ψ is $S_1 \times S_2$. Intuitively, a variable (s_1, s_2) stands for the proposition that state s_1 can be simulated by state s_2.

The clauses of Ψ encode constraints imposed by a simulation relation and are constructed as follows. For each $s_1 \in S_1$, each $s_2 \in S_2$, each $\alpha \in \Sigma$, and each $s'_1 \in Succ(s_1, \alpha)$, we add the following clause to Ψ: $(s_1, s_2) \Rightarrow \bigvee_{s'_2 \in Succ(s_2,\alpha)} (s'_1, s'_2)$. Intuitively, the above clause expresses the requirement that for s_2 to simulate s_1, at least one α-successor of s_2 must simulate s'_1. Also, for each $s_1 \in Init_1$, we add the following clause to Ψ: $\bigvee_{s_2 \in Init_2} (s_1, s_2)$. These clauses assert that every initial state of M_1 must be simulated by some initial state of M_2. Now, Ψ has the following simple property. Let X be any satisfying assignment of Ψ, and for any variable $v = (s_1, s_2)$, let us write $X(s_1, s_2)$ to mean the Boolean value assigned to v by X. Then, the relation $\mathcal{R} = \{(s_1, s_2) \mid X(s_1, s_2) = \text{TRUE}\}$ is a simulation relation between M_1 and M_2.

Therefore, we can construct a minimal simulation between M_1 and M_2 by constructing Ψ and then looking for a satisfying assignment X such that the number of variables assigned TRUE by X is as small as possible. This task can be achieved by using a solver for pseudo-Boolean formulas [Aloul 02]. A pseudo-Boolean formula is essentially a propositional formula coupled with an arithmetic constraint over the propositional variables (where TRUE is treated as one and FALSE as zero). More specifically, recall that the set of variables of Ψ is $S_1 \times S_2$. We thus solve for Ψ along with the constraint that the following sum be minimized: $\Upsilon = \sum_{s \in S_1 \times S_2} s$. We then construct a minimal simulation relation using any satisfying assignment to Ψ that also minimizes Υ.

Hardness of Finding Minimal Simulation Relations. One may complain that solving pseudo-Boolean formula satisfiability (an NP-complete problem) to verify simulation (for which polynomial time algorithms exist) is overkill. However, the use of a pseudo-Boolean solver is justified by the fact that finding a *minimal* simulation between two finite LTSs is actually an NP-hard problem.

We now prove this claim by reducing *subgraph isomorphism*, a well-known NP-complete problem, to the problem of finding a minimal simulation relation between two LTSs. In the rest of this section, whenever we mention a simulation relation between the LTSs M_1 and M_2, we also tacitly assume that every initial state of M_1 is simulated by some initial state of M_2.

Definition 12 (Graph). *An undirected graph is a pair (V, E) where V is a set of vertices and $E \subseteq V \times V$ is a symmetric irreflexive relation denoting edges.*

Definition 13 (Subgraph Isomorphism). *Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ such that $|V_1| < |V_2|$, we say that G_1 is subgraph isomorphic to G_2 iff there exists an injection $\mu : V_1 \to V_2$ that obeys the following condition:*
$\forall v \in V_1 \centerdot \forall v' \in V_1 \centerdot (v, v') \in E_1 \iff (\mu(v), \mu(v')) \in E_2$.

Note that we do not allow self-loops in graphs. It is well-known that, given two arbitrary graphs G_1 and G_2, the problem of deciding whether G_1 is subgraph isomorphic to G_2 is NP-complete. We now show that this problem has a log-space reduction to the problem of finding a minimal simulation relation between two LTSs. In essence, from G_1 and G_2, we

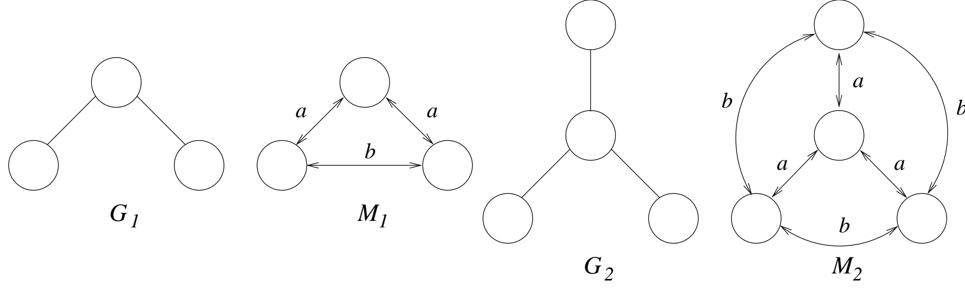

Figure 2: Example Graphs and LTSs Constructed from Them

construct the LTSs M_1 and M_2 such that G_1 is subgraph isomorphic to G_2 iff a minimal simulation relation between M_1 and M_2 has the same size as G_1.

Recall that $G_1 = (V_1, E_1)$. We construct $M_1 = (S_1, Init_1, \Sigma, T_1)$ as follows: (1) the states of M_1 are exactly the vertices of V_1, i.e., $S_1 = V_1$, (2) all states of M_1 are initial (i.e., $Init_1 = S_1$), (3) M_1 has two actions a and b (i.e., $\Sigma = \{a, b\}$), and (4) the transitions T_1 of M_1 are set up as follows: (1) for each $(v, v') \in E_1$ we add $v \xrightarrow{a} v'$ and $v' \xrightarrow{a} v$ to T_1 and (2) for each $(v, v') \notin E_1$ we add $v \xrightarrow{b} v'$ and $v' \xrightarrow{b} v$ to T_1. The LTS M_2 is constructed from graph G_2 in an analogous manner. As an example, Figure 2 shows two graphs G_1 and G_2, as well as the LTSs M_1 and M_2 constructed from them. A bidirectional arrow between two states (of M_1 or M_2) represents a pair of transitions—one from each state to the other. Note that M_1 and M_2 can be constructed using logarithmic additional space. Now our NP-hardness reduction is completed by the following theorem.

Theorem 5 (Reduction). *Let n be the number of states of M_1 (i.e., $n = |S_1|$). Then, G_1 is subgraph isomorphic to G_2 iff a minimal simulation relation between M_1 and M_2 has n elements.*

Proof. Let $\mathcal{R}$ be any minimal simulation relation between M_1 and M_2. First, note that since every state of M_1 is initial, $\mathcal{R}$ must have at least n elements. To prove the forward implication, assume that G_1 is subgraph isomorphic to G_2, and let μ be a function satisfying the condition of Definition 13. Then, the following relation $\mathcal{R}$ is clearly a minimal simulation relation, since it has exactly n elements:

$$\mathcal{R} = \{(v, \mu(v)) \mid v \in S_1\}$$

To prove the reverse implication, suppose that $\mathcal{R}$ is a minimal simulation relation between M_1 and M_2 containing n elements. Since each of the n states of M_1 must be simulated, $\mathcal{R}$ must relate each state of M_1 to a unique state of M_2. Now, consider the function $\mu : S_1 \rightarrow S_2$, which is defined as follows:

$$\mu(v_1) = \text{unique } v_2 \in S_2 \text{ such that } (v_1, v_2) \in \mathcal{R}$$

We show that μ satisfies the criterion in Definition 13 as follows:

$$(v, v') \in E_1 \Rightarrow v \xrightarrow{a} v' \Rightarrow \mu(v) \xrightarrow{a} \mu(v') \Rightarrow (\mu(v), \mu(v')) \in E_2$$

$$(v, v') \notin E_1 \Rightarrow v \xrightarrow{b} v' \Rightarrow \mu(v) \xrightarrow{b} \mu(v') \Rightarrow (\mu(v), \mu(v')) \notin E_2$$

Now, we must show that μ is an injection. Suppose that μ was not an injection. In that case, we have two elements v_1 and v'_1 of V_1 such that $\mu(v_1) = \mu(v'_1) = v_2$, let's say. But then, M_2 must contain at least one of the following two transitions: $v_2 \xrightarrow{a} v_2$ or $v_2 \xrightarrow{b} v_2$. This is clearly impossible, and it also completes our proof. □

7 Experimental Results

We implemented our techniques in ComFoRT [Chaki 05b] and experimented with a set of Linux and Windows NT device drivers, OpenSSL, and the Micro-C operating system. All our experiments were carried out on a dual Intel Xeon 2.4 GHz machine with 4 GB RAM running Redhat 9. Our results are summarized in Figure 3, which obeys the following convention.

Name	*LOC*	CVC	VAMPYRE	*SAT*	*Cert*	*Core*	*Improve*
ide.c (safe)	7428	80720	×	**100**	**703**	>2000	807
ide.c (live)	7428	82653	×	**100**	**1319**	>2000	827
tlan.c (safe)	6523	11145980	×	**517**	**4663**	>200	21559
tlan.c (live)	6523	90155057	×	**572**	**74281**	>200	157614
aha152x.c (safe)	10069	247435	×	**210**	**2102**	>1500	1178
aha152x.c (live)	10069	247718	×	**210**	**3968**	>1500	1180
synclink.c (safe)	17104	9822	×	**53**	**185**	>500	185
synclink.c (live)	17104	9862	×	**53**	**327**	>500	186
hooks.c (safe)	30923	597642	×	**369**	**2004**	>1500	1629
hooks.c (live)	30923	601175	×	**368**	**3102**	>1500	1624
cdaudio.c (safe)	17798	248915	156787*	**209**	**2006**	>1000	750
diskperf.c (safe)	4824	117172	×	**106**	**955**	>2500	1105
floppy.c (safe)	17386	451085	60129*	**318**	**2595**	>3000	189
kbfiltr.c (safe)	12131	56682	7619*	**51**	**528**	>2500	149
parclass.c (safe)	26623	460973	×	**262**	**2156**	>4500	1759
parport.c (safe)	61781	2278120	102967*	**529**	**3568**	>5000	195
SSL-srvr (simul)	2483	1287290	19916	**261**	**1055**	>150	76
SSL-clnt (simul)	2484	189401	27189	**155**	**740**	>200	175
Micro-C (safe)	6272	416930	118162	**262**	**2694**	>5500	451
Micro-C (live)	6272	435450	×	**263**	**7571**	>5500	1656

Figure 3: Comparison of CVC, VAMPYRE, *and SAT-Based Proof Generation*

The symbol × indicates that results are not available. Best figures appear in boldface. The *LOC* column contains lines of code. The CVC, VAMPYRE, and *SAT* columns refer to proof sizes in bytes (after compressing with the `gzip` utility) obtained with CVC, VAMPYRE, and SAT, respectively. The CVC statistics were obtained via ComFoRT. The BLAST statistics were obtained using either Version 2.0 of BLAST or an existing publication (indicated by an asterisk [*]). The *Cert* column mentions the gzipped certificate (i.e., witness + proof of the verification condition) size with SAT. The *Core* column contains the factors by which the unsatisfiable core is smaller than the original SAT formula. Finally, the *Improve* column refers to factors by which SAT-based proofs are smaller than the nearest other proofs.

The Linux device drivers were obtained from kernel 2.6.11.10. We checked that the drivers obey the following conventions with `spin_lock` and `spin_unlock`: (1) locks must be acquired and released alternately beginning with an acquire (*safe*) and (2) every acquire must be eventually followed by a release (*live*). The Windows drivers are instrumented so that an `ERROR` location is reached if any illegal behavior is executed. We certified that `ERROR` is

unreachable for all the drivers we experimented with. For OpenSSL (Version 0.9.6c), we certified that the initial handshake between a server and a client obeys the protocol specified in the SSL 3.0 specification. For Micro-C (Version 2.72), we certified that the calls to `OS_ENTER_CRITICAL` and `OS_EXIT_CRITICAL` obey the two locking conventions mentioned above.

In almost all cases, SAT-based proofs are over 100 times more compact than those generated by CVC and VAMPYRE. In one instance—*tlan.c (live)*—the improvement is by a factor of more than 10^5. We also find that an important reason for such improvement is that the UNSAT-cores are much smaller (by over two to three orders of magnitude) than the actual SAT formulas. Upon closer inspection, we discovered that this is due to the simplicity of the verification conditions. For instance, the device drivers satisfy the locking conventions because of local coding conventions (every procedure with a lock has a matching unlock). In practice, this results in very simple verification conditions. Proofs generated by CVC and VAMPYRE suffer from redundancies and inefficient encodings and therefore turn out to be large even for such simple formulas. In contrast, SAT formulas generated from these simple verification conditions are characterized by small unsatisfiable cores.

We note that the total size of the certificate is usually dominated by the size of the witness. Finally, we find that certificates for liveness policies tend to be larger than those for the corresponding safety policies. This is due to the additional information required to encode the ranking function, which is considerably more complex for liveness specifications.

8 Conclusion

We have formalized a notion of witnesses for satisfaction of linear temporal logic specifications by infinite state programs and of simulation conformance between infinite state programs and finite state machine specifications. We have described how such witnesses may be constructed via predicate abstraction and validated by generating and proving verification conditions.

We have proposed the use of SAT-based theorem provers and resolution proofs in proving these verification conditions. Our experimental results on nontrivial benchmarks suggest that a SAT-based approach can yield extremely compact proofs. Our SAT-based approach also overcomes several limitations of conventional theorem provers when applied to the verification of real-life programs.

There is an evident need for techniques to obtain compact proofs in a wide variety of disciplines. Algorithms to compress and compact proof representations are currently a topic of active research. In this context, the use of powerful SAT technology appears to be a very promising idea and warrants further investigation. Extending the set of properties that can be certified effectively would also enhance the scope of the work presented in this report. Finally, the usefulness of SAT for constructing compact proofs for the purpose of generating proof-carrying binary code remains an important yet open question.

We are grateful to Stephen Magill, Aleksandar Nanevski, Peter Lee, and Edmund Clarke for their insight on Proof-Carrying Code and model checking. We also thank Rupak Majumdar and Ranjit Jhala for providing us with the Windows driver benchmarks and Anubhav Gupta for his advice on using ZCHAFF.

References

[Aloul 02] Aloul, F.; Ramani, A.; Markov, I.; & Sakallah, K. "PBS: A Backtrack Search Pseudo-Boolean Solver", 346–353. *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT '02).* Cincinnati, OH, May 6–9, 2002. Informal proceedings.

[Appel 01] Appel, A. W. "Foundational Proof-Carrying Code", 247–258. *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01).* Boston, MA, June 16–19, 2001. Los Alamitos, CA: IEEE Computer Society Press, 2001.

[Arons 01] Arons, T.; Pnueli, A.; Ruah, S.; Xu, J.; & Zuck, L. D. "Parameterized Verification with Automatically Computed Inductive Assertions", 221–234. *Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01)*, Volume 2102 of Lecture Notes in Computer Science. Paris, France, July 18–22, 2001. New York, NY: Springer-Verlag, 2001.

[Balaban 05] Balaban, I.; Pnueli, A.; & Zuck, L. D. "Shape Analysis by Predicate Abstraction", 164–180. *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '05)*, Volume 3385 of Lecture Notes in Computer Science. Paris, France, January 17–19, 2005. New York, NY: Springer-Verlag, 2005.

[Ball 01] Ball, T. & Rajamani, S. K. "Automatically Validating Temporal Safety Properties of Interfaces", 103–122. *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN '01)*, Volume 2057 of Lecture Notes in Computer Science. Toronto, Canada, May 19–20, 2001. New York, NY: Springer-Verlag, 2001.

[Bernard 02] Bernard, A. & Lee, P. "Temporal Logic for Proof-Carrying Code", 31–46. *Proceedings of the 18th International Conference on Automated Deduction (CADE '02)*, Volume 2392 of Lecture Notes in Computer Science. Copenhagen, Denmark, July 27–30, 2002. New York, NY: Springer-Verlag, 2002.

[Chaki 04a] Chaki, S.; Clarke, E.; Groce, A.; Jha, S.; & Veith, H. "Modular Verification of Software Components in C". *IEEE Transactions on Software Engineering (TSE) 30*, 6 (June 2004): 388–402.

[Chaki 04b] Chaki, S.; Clarke, E. M.; Ouaknine, J.; Sharygina, N.; & Sinha, N. "State/Event-Based Software Model Checking", 128–147. *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM '04)*, Volume 2999 of Lecture Notes in Computer Science. Canterbury, UK, April 4–7, 2004. New York, NY: Springer-Verlag, 2004.

[Chaki 05a] Chaki, S.; Clarke, E.; Jha, S.; & Veith, H. "An Iterative Framework for Simulation Conformance". *Journal of Logic and Computation (JLC) 15*, 4 (August 2005): 465–488.

[Chaki 05b] Chaki, S.; Ivers, J.; Sharygina, N.; & Wallnau, K. "The ComFoRT Reasoning Framework", 164–169. *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, Volume 3576 of Lecture Notes in Computer Science. Edinburgh, Scotland, July 6–10, 2005. New York, NY: Springer-Verlag, 2005.

[Clarke 82] Clarke, E. & Emerson, A. "Design and Synthesis of Synchronization Skeletons for Branching Time Temporal Logic", 52–71. *Proceedings of Workshop on Logic of Programs*, Volume 131 of Lecture Notes in Computer Science. Yorktown Heights, New York, May 4–6, 1982. Berlin, Germany: Springer-Verlag, 1982.

[Clarke 00] Clarke, E. M.; Grumberg, O.; & Peled, D. *Model Checking*. Cambridge, MA: MIT Press, 2000.

[Clarke 03] Clarke, E. M.; Grumberg, O.; Jha, S.; Lu, Y.; & Veith, H. "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking". *Journal of the ACM (JACM) 50*, 5 (September 2003): 752–794.

[Clarke 04] Clarke, E.; Kroening, D.; & Lerda, F. "A Tool for Checking ANSI-C Programs", 168–176. *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, Volume 2988 of Lecture Notes in Computer Science. Barcelona, Spain, March 29–April 2, 2004. New York, NY: Springer-Verlag, 2004.

[Cook 05a] Cook, B.; Kroening, D.; & Sharygina, N. "Symbolic Model Checking for Asynchronous Boolean Programs", 75–90. *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN '05)*, Volume 3639 of Lecture Notes in Computer Science. San Francisco, CA, August 22–24, 2005. New York, NY: Springer-Verlag, 2005.

[Cook 05b] Cook, B.; Podelski, A.; & Rybalchenko, A. "Abstraction Refinement for Termination", 87–101. *Proceedings of the 12th International Static Analysis Symposium (SAS '05)*, Volume 3672 of Lecture Notes in Computer Science. London, UK, September 7–9, 2005. New York, NY: Springer-Verlag, 2005.

[Graf 97] Graf, S. & Saïdi, H. “Construction of Abstract State Graphs with PVS”, 72–83. *Proceedings of the 9th International Conference on Computer Aided Verification (CAV ’97)*, Volume 1254 of Lecture Notes in Computer Science. Haifa, Israel, June 22–25, 1997. New York, NY: Springer-Verlag, 1997.

[Hamid 02] Hamid, N. A.; Shao, Z.; Trifonov, V.; Monnier, S.; & Ni, Z. “A Syntactic Approach to Foundational Proof-Carrying Code”, 89–100. *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS ’02).* Copenhagen, Denmark, July 22–25, 2002. Los Alamitos, CA: IEEE Computer Society Press, 2002.

[Henzinger 02a] Henzinger, T. A.; Jhala, R.; Majumdar, R.; Necula, G. C.; Sutre, G.; & Weimer, W. “Temporal-Safety Proofs for Systems Code”, 526–538. *Proceedings of the 14th International Conference on Computer Aided Verification (CAV ’02)*, Volume 2404 of Lecture Notes in Computer Science. Copenhagen, Denmark, July 27–31, 2002. New York, NY: Springer-Verlag, 2002.

[Henzinger 02b] Henzinger, T. A.; Jhala, R.; Majumdar, R.; & Sutre, G. “Lazy Abstraction”, 58–70. *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL ’02)*, Volume 37(1) of *SIGPLAN Notices*. Portland, OR, January 16–18, 2002. New York, NY: ACM Press, 2002.

[Kroening 02] Kroening, D. “Application Specific Higher Order Logic Theorem Proving”, 5–15. *Proceedings of the Verification Workshop (VERIFY’02).* Copenhagen, Denmark, July 25–26, 2002. Copenhagen, Denmark: Department of Computer Science, University of Copenhagen (DIKU), 2002.

[Kupferman 04] Kupferman, O. & Vardi, M. Y. “From Complementation to Certification”, 591–606. *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS ’04)*, Volume 2988 of Lecture Notes in Computer Science. Barcelona, Spain, March 29–April 2, 2004. New York, NY: Springer-Verlag, 2004.

[Michael 00] Michael, N. G. & Appel, A. W. “Machine Instruction Syntax and Semantics in Higher Order Logic”, 7–24. *Proceedings of the 17th International Conference on Automated Deduction (CADE ’00)*, Volume 1831 of Lecture Notes in Computer Science. Pittsburgh, PA, June 17–20, 2000. New York, NY: Springer-Verlag, 2000.

[Moskewicz 01] Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; & Malik, S. “Chaff: Engineering an Efficient SAT Solver”, 530–535. *Proceedings of the 38th ACM IEEE Design Automation Conference (DAC ’01).* Las Vegas, Nevada, June 18–22, 2001. New York, NY: ACM Press, 2001.

[Namjoshi 01] Namjoshi, K. S. "Certifying Model Checkers", 2–13. *Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01)*, Volume 2102 of Lecture Notes in Computer Science. Paris, France, July 18–22, 2001. New York, NY: Springer-Verlag, 2001.

[Namjoshi 03] Namjoshi, K. S. "Lifting Temporal Proofs through Abstractions", 174–188. *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '03)*, Volume 2575 of Lecture Notes in Computer Science. New York, NY, January 9–11, 2002. New York, NY: Springer-Verlag, 2003.

[Necula 96] Necula, G. C. & Lee, P. "Safe Kernel Extensions without Runtime Checking", 229–243. *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation (OSDI '96).* Seattle, WA, October 28–31, 1996. New York, NY: ACM Press, 1996.

[Necula 97] Necula, G. C. "Proof-Carrying Code", 106–119. *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97).* Paris, France, January 15–17, 1997. New York, NY: ACM Press, 1997.

[Necula 98a] Necula, G. C. & Lee, P. "Efficient Representation and Validation of Proofs", 93–104. *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS '98).* Indianapolis, IN, June 21–24, 1998. Los Alamitos, CA: IEEE Computer Society Press, 1998.

[Necula 98b] Necula, G. C. & Lee, P. "Safe, Untrusted Agents Using Proof-Carrying Code", 61–91. *Proceedings of Mobile Agents and Security*, Volume 1419 of Lecture Notes in Computer Science. New York, NY: Springer-Verlag, 1998.

[Necula 01] Necula, G. C. & Rahul, S. P. "Oracle-Based Checking of Untrusted Software", 142–154. *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '01)*, Volume 36(3) of *SIGPLAN Notices.* London, UK, January 17–19, 2001, New York, NY: ACM Press, 2001.

[Nelson 80] Nelson, G. "Techniques for Program Verification". PhD diss., Stanford University, California, 1980.

[Zhang 03] Zhang, L. & Malik, S. "Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications", 10880–10885. *Proceedings of 2003 Design, Automation, and Test in Europe Conference and Exposition (DATE 2003).* Munich, Germany, March 3–7, 2003. Los Alamitos, CA: IEEE Computer Society Press, 2003.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	February 2006	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
SAT-Based Software Certification	FA8721-05-C-0003

6. AUTHOR(S)

Sagar Chaki

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

CMU/SEI-2006-TN-004

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This report formalizes a notion of witnesses as the basis of certifying the correctness of software. The first part of the report is concerned with witnesses for the satisfaction of linear temporal logic specifications by infinite state programs and shows how such witnesses may be constructed via predicate abstraction and validated by generating and proving verification conditions. In addition, the first part of this report proposes the use of theorem provers based on Boolean propositional satisfiability (SAT) and resolution proofs in validating these verification conditions. In addition to yielding extremely compact proofs, a SAT-based approach overcomes several limitations of conventional theorem provers when applied to the verification of programs written in real-life programming languages.

The second part of this report formalizes a notion of witnesses of simulation conformance between infinite state programs and finite state machine specifications. The report also proves that computing a minimal simulation relation between two finite state machines is an NP-hard problem. Finally, the report presents algorithms to construct simulation witnesses of minimal size by solving pseudo-Boolean constraints. The author's experiments on several nontrivial benchmarks suggest that a SAT-based approach can yield extremely compact proofs—in some cases by a factor of over 10^5—when compared to existing non-SAT-based theorem provers.

14. SUBJECT TERMS	15. NUMBER OF PAGES
software certification, Boolean satisfiability, proofs, ranking functions, simulation	41

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL